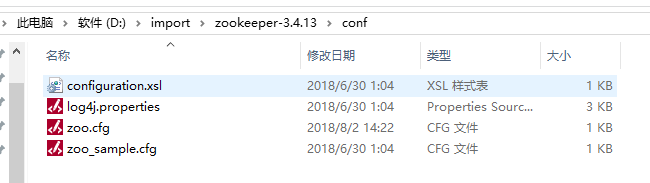
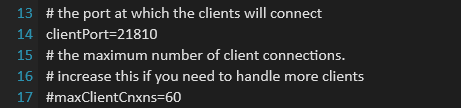
安装

1.从网上下载,注意beta 为测试版,为了追求稳定选择稳定版

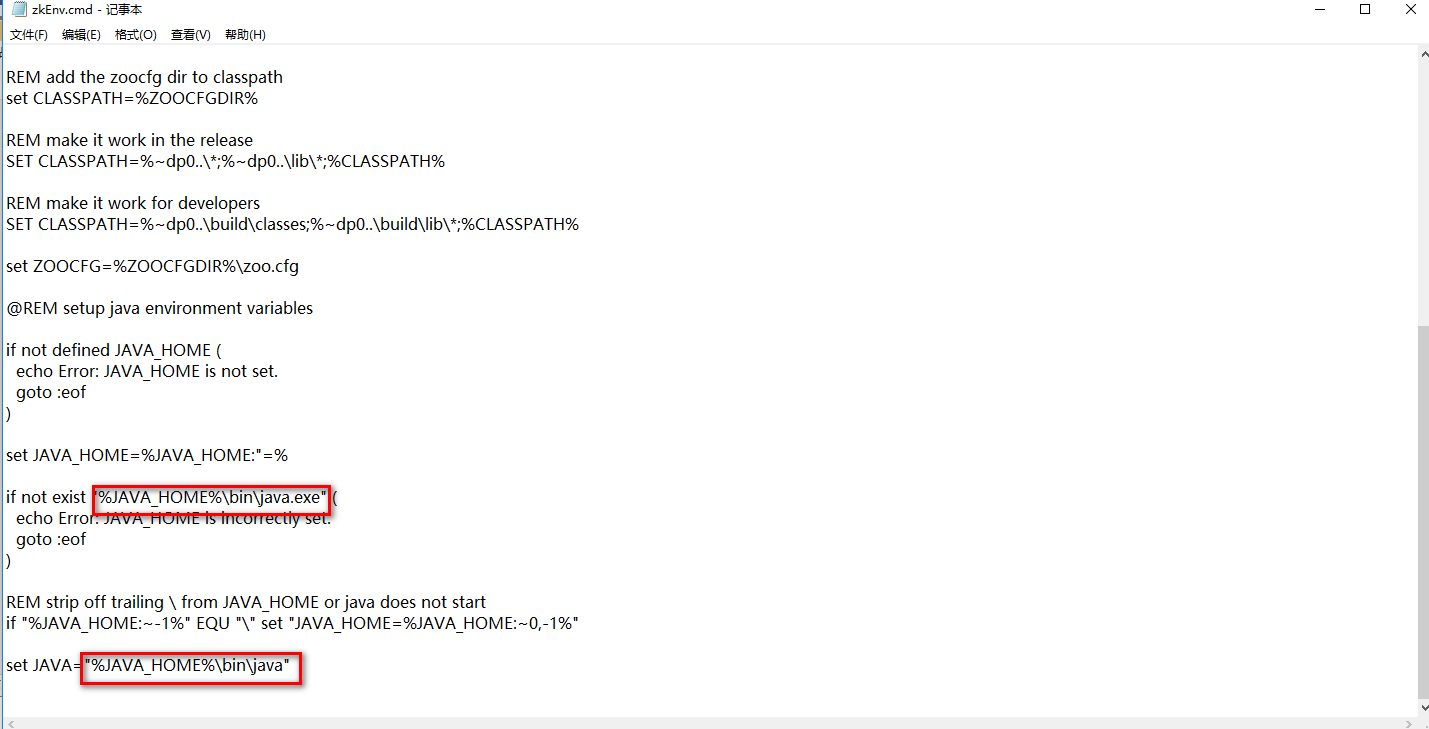
2.解压完毕后,注意有以下操作:



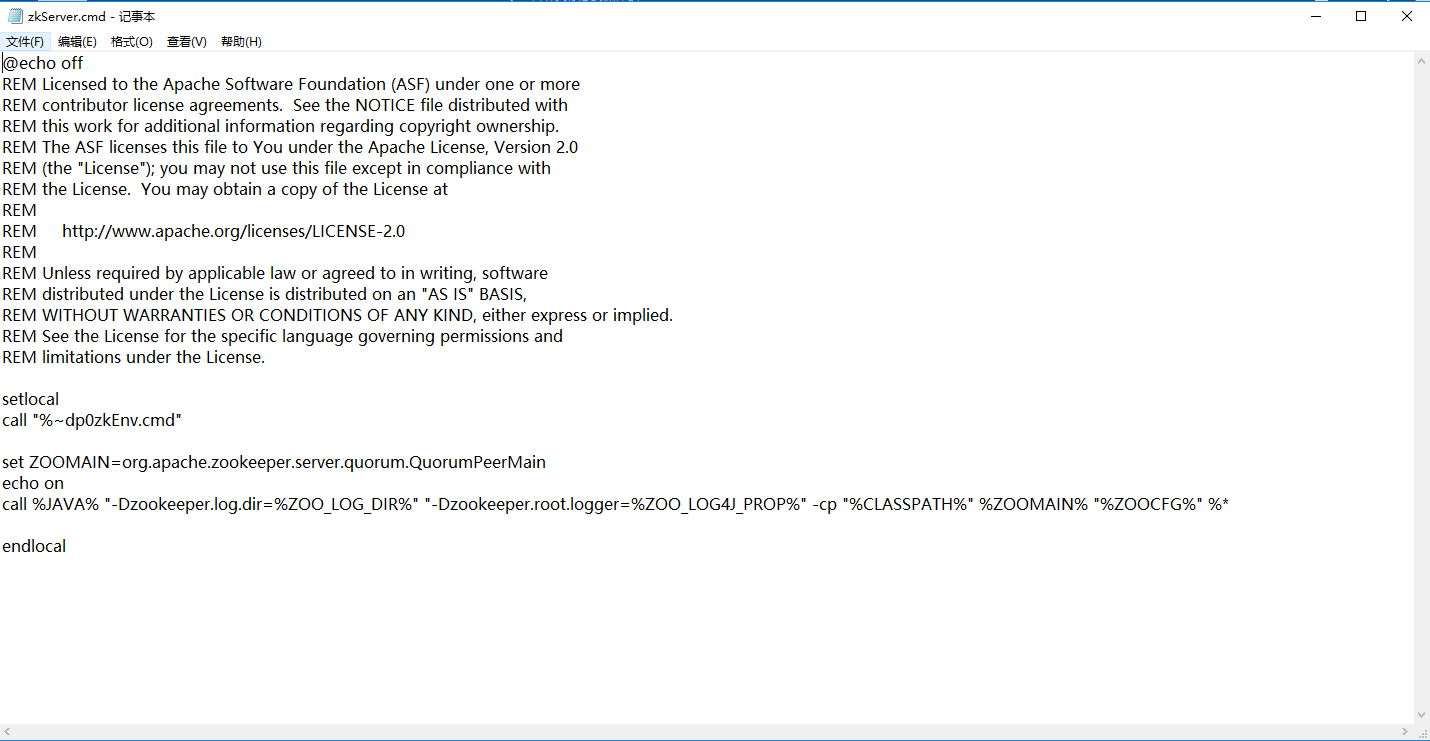
Zoo.cfg为zoo\_sample.cfg的副本,里面的端口2180如果被占用是可以修改的



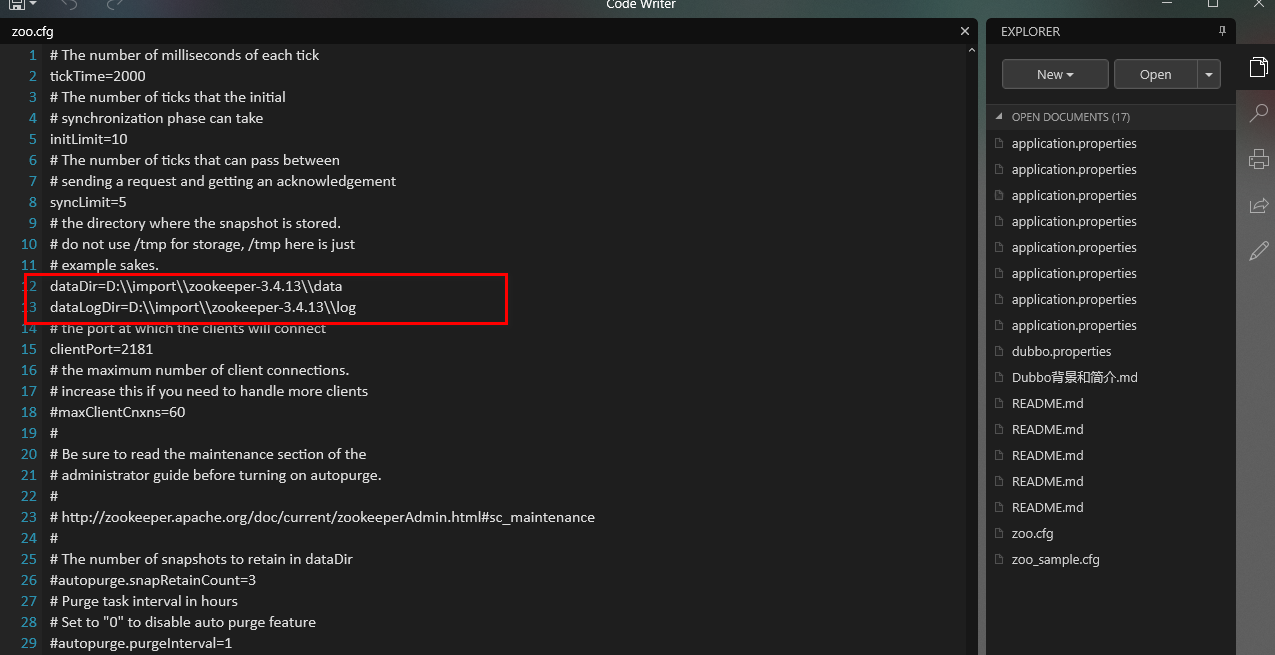
3.另外在进入bin目录后启动如果报错可以更改以下文件:



如果出现了一闪而过的情况可酌情在以下文件中添加 pause (暂停)

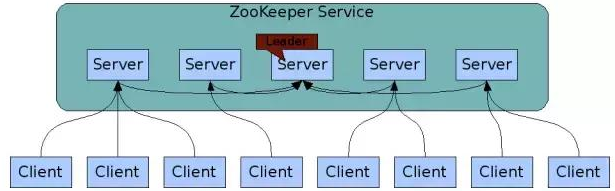


还要注意在目录下创建data与log两个文件夹,更改zoo.cfg



## Zookeeper 介绍

Zookeeper 是用集群管理者,用于高度分布式协同调度,管理,发布管理接口的服务. 是Google的Chubby一个开源的实现.底层仅仅提供监听,监听用户所提交的请求.可参考图例加深理解.也可以提供选举,这里的选举不是自身的选举,而是提供选举接口.



特性:

1. [文件系统](#_Zookeeper文件系统)
2. [通知机制](#_通知系统)
3. 故障转移
4. 权限ACL

提供功能:

1. [命名服务](#_命名服务)
2. [配置管理](#_配置管理)
3. [集群管理](#_集群管理)
4. [分布式锁](#_分布式锁)
5. [队列管理](#_队列管理.)

### 文件系统

Zookeeper 能像windows系统文件目录一样,以节点的形式管理接口.

节点通常分为:

1. 临时节点
2. 临时顺序节点
3. 持久化节点
4. 持久化临时节点

临时节点（EPHEMERAL）:

临时创建的节点,需要时创建,当会话完毕后就会自动删除这个节点,当然也可以手动删除,且这个节点没有子节点,客户端连接不关闭就会一直存在.

临时顺序节点（EPHEMERAL\_SEQUENTIAL）:

拥有跟临时节点一样的特性,但相交临时节点来说,创建节点的时候也会生成一个序号.仅次而已.这个序号在[分布式锁](#_分布式锁)中会用到.

持久化节点（PERSISTENT）:

该节点创建后会持续的存在于zookeeper中,非手动删除会一直存在.

持久化顺序节点（PERSISTENT\_SEQUENTIAL）:

和持久节点一样,但创建时与临时顺序一样,会产生一个序列号.这个序列号是10位的,且该节点的父节点是唯一的.如果序列号大于2^32次方就会溢出.

注意:节点的任意更改都会版本号增加, 有三个版本号dataversion（数据版本号）、cversion（子节点版本号）、aclversion（节点所拥有的ACL版本号）.

节点的改变同时会产生一个zxid 的时间戳,这个zxid是一个唯一的事务id.

同时Zookeeper 不会缓存已经断掉的节点

### 通知系统

客户端节点上设置Watches(监控器) 当节点状态发生变化时(比如,节点的CRUD),会自动触发对应操作.如果服务器上的节点发生改变也会在第一时间通过监控器机通知客户端.

### 故障转移

传统的是通过心跳监听机制,来监听服务而Zookeeper是通过监控,一旦节点出现问题就会从Zookeeper中删除.通过临时节点和监控可以实现故障转移.

### 命名服务

在zookeeper的系统中会创建一个path目录,此目录是唯一的.由于服务接口存在于服务器上而Zookeeper仅是对他们的统一管理,一旦接口过多如果就很有可能出现隐患,仅通过上下的关系所暴露的名称是不足以找到所对应接口实际地址.所以通过命名服务部署上下所约定好的path极为重要.

场景：有一组服务器向客户端提供某种服务（例如：我前面做的分布式网站的服务端，就是由四台服务器组成的集群，向前端集群提供服务），我们希望客户端每次请求服务端都可以找到服务端集群中某一台服务器，这样服务端就可以向客户端提供客户端所需的服务。对于这种场景，我们的程序中一定有一份这组服务器的列表，每次客户端请求时候，都是从这份列表里读取这份服务器列表。那么这分列表显然不能存储在一台单节点的服务器上，否则这个节点挂掉了，整个集群都会发生故障，我们希望这份列表时高可用的。高可用的解决方案是：这份列表是分布式存储的，它是由存储这份列表的服务器共同管理的，如果存储列表里的某台服务器坏掉了，其他服务器马上可以替代坏掉的服务器，并且可以把坏掉的服务器从列表里删除掉，让故障服务器退出整个集群的运行，而这一切的操作又不会由故障的服务器来操作，而是集群里正常的服务器来完成。这是一种主动的分布式数据结构，能够在外部情况发生变化时候主动修改数据项状态的数据机构。Zookeeper框架提供了这种服务。这种服务名字就是：统一命名服务，它和javaEE里的JNDI服务很像。

### 配置管理/协同配置

就是对所有在Zookeeper中存在的资源进行管理,同时启动监控,一旦变化,快速通知.

场景：配置管理。在分布式系统里，我们会把一个服务应用分别部署到n台服务器上，这些服务器的配置文件是相同的（例如：我设计的分布式网站框架里，服务端就有4台服务器，4台服务器上的程序都是一样，配置文件都是一样），如果配置文件的配置选项发生变化，那么我们就得一个个去改这些配置文件，如果我们需要改的服务器比较少，这些操作还不是太麻烦，如果我们分布式的服务器特别多，比如某些大型互联网公司的hadoop集群有数千台服务器，那么更改配置选项就是一件麻烦而且危险的事情。这时候zookeeper就可以派上用场了，我们可以把zookeeper当成一个高可用的配置存储器，把这样的事情交给zookeeper进行管理，我们将集群的配置文件拷贝到zookeeper的文件系统的某个节点上，然后用zookeeper监控所有分布式系统里配置文件的状态，一旦发现有配置文件发生了变化，每台服务器都会收到zookeeper的通知，让每台服务器同步zookeeper里的配置文件，zookeeper服务也会保证同步操作原子性，确保每个服务器的配置文件都能被正确的更新。

### 集群管理

所谓的集群管理就是将单个处理,变成团队处理,这样在提供同一服务.具体派遣谁来做,又是一个问题.根据这个问题就引入到另一概念”大哥”.”大哥”会去管理.这个管理如新收到的”小弟”带大家认识下,有问题的”小弟”处理下等.当我们在调取后续服务的时候会把分配的任务给这个”大哥”,由这个”总管”去处理后续操作.

原理：zookeeper在配置文件中并没有指定master和slave，但是，zookeeper在工作时，只有一个节点为leader，其余节点为follower，leader是通过内部的选举机制临时产生的。

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

当leader(“大哥”)崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。先介绍basic paxos流程：

1 .选举线程由当前Server发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

2 .选举线程首先向所有Server发起一次询问(包括自己)；

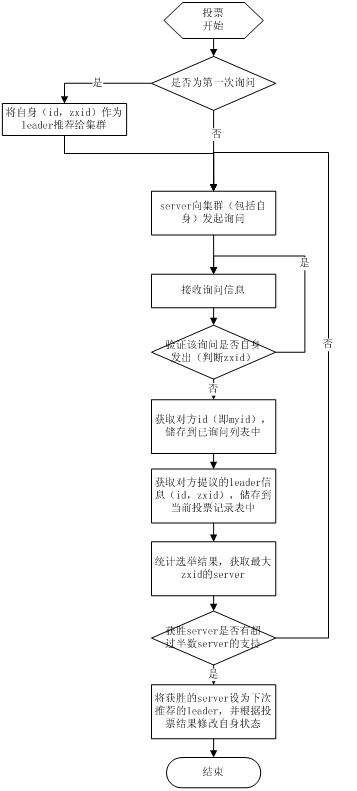
3 .选举线程收到回复后，验证是否是自己发起的询问(验证zxid是否一致)，然后获取对方的id(myid)，并存储到当前询问对象列表中，最后获取对方提议的leader相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

4.  收到所有Server回复以后，就计算出zxid最大的那个Server，并将这个Server相关信息设置成下一次要投票的Server；

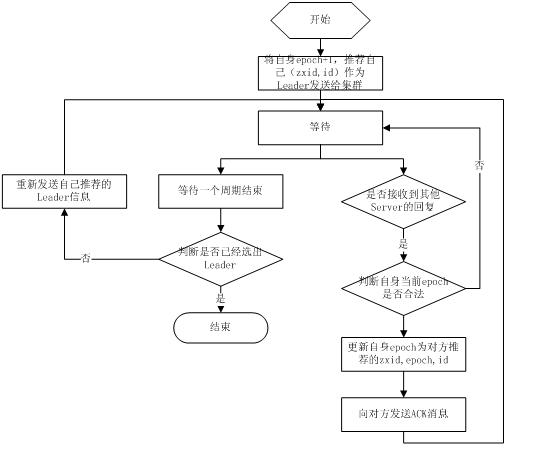
5.  线程将当前zxid最大的Server设置为当前Server要推荐的Leader，如果此时获胜的Server获得n/2 + 1的Server票数， 设置当前推荐的leader为获胜的Server，将根据获胜的Server相关信息设置自己的状态，否则，继续这个过程，直到leader被选举出来。

通过流程分析我们可以得出：要使Leader获得多数Server的支持，则Server总数必须是奇数2n+1，且存活的Server的数目不得少于n+1.

每个Server启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的server还会从磁盘快照中恢复数据和会话信息，zk会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。选主的具体流程图如下所示：

[](http://static.oschina.net/uploads/img/201308/08171345_J3LF.jpg)

fast paxos流程是在选举过程中，某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。其流程图如下所示：

[](http://static.oschina.net/uploads/img/201308/08171346_zLlp.jpg)

同步流程

选完leader以后，zk就进入状态同步过程。

1. leader等待server连接；

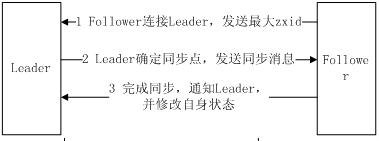
2 .Follower连接leader，将最大的zxid发送给leader；

3 .Leader根据follower的zxid确定同步点；

4 .完成同步后通知follower 已经成为uptodate状态；

5 .Follower收到uptodate消息后，又可以重新接受client的请求进行服务了。

流程图如下所示：

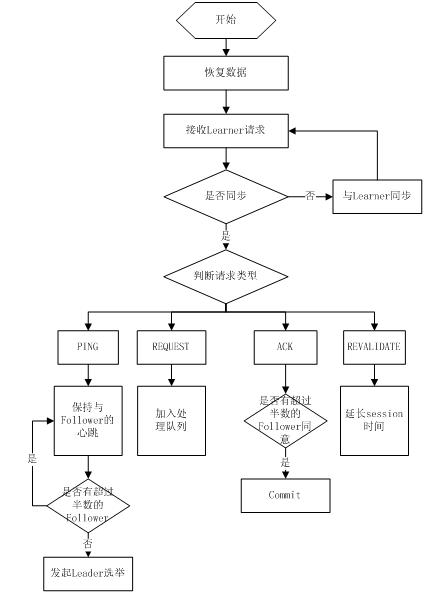
[](http://static.oschina.net/uploads/img/201308/08171346_oExa.jpg)

Leader主要有三个功能：

1 .恢复数据；

2 .维持与Learner的心跳，接收Learner请求并判断Learner的请求消息类型；

3 .Learner的消息类型主要有PING消息、REQUEST消息、ACK消息、REVALIDATE消息，根据不同的消息类型，进行不同的处理。

PING消息是指Learner的心跳信息；REQUEST消息是Follower发送的提议信息，包括写请求及同步请求；ACK消息是Follower的对提议的回复，超过半数的Follower通过，则commit该提议；REVALIDATE消息是用来延长SESSION有效时间。  
Leader的工作流程简图如下所示，在实际实现中，流程要比下图复杂得多，启动了三个线程来实现功能。[](http://static.oschina.net/uploads/img/201308/08171346_87iA.jpg)

Follower工作流程

Follower主要有四个功能：

1. 向Leader发送请求（PING消息、REQUEST消息、ACK消息、REVALIDATE消息）；

2 .接收Leader消息并进行处理；

3 .接收Client的请求，如果为写请求，发送给Leader进行投票；

4 .返回Client结果。

Follower的消息循环处理如下几种来自Leader的消息：

1 .PING消息： 心跳消息；

2 .PROPOSAL消息：Leader发起的提案，要求Follower投票；

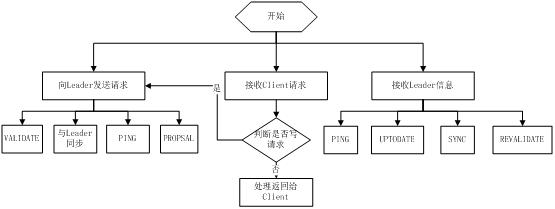
3 .COMMIT消息：服务器端最新一次提案的信息；

4 .UPTODATE消息：表明同步完成；

5 .REVALIDATE消息：根据Leader的REVALIDATE结果，关闭待revalidate的session还是允许其接受消息；

6 .SYNC消息：返回SYNC结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

Follower的工作流程简图如下所示，在实际实现中，Follower是通过5个线程来实现功能的。

[](http://static.oschina.net/uploads/img/201308/08171346_slOJ.jpg)

对于observer的流程不再叙述，observer流程和Follower的唯一不同的地方就是observer不会参加leader发起的投票。

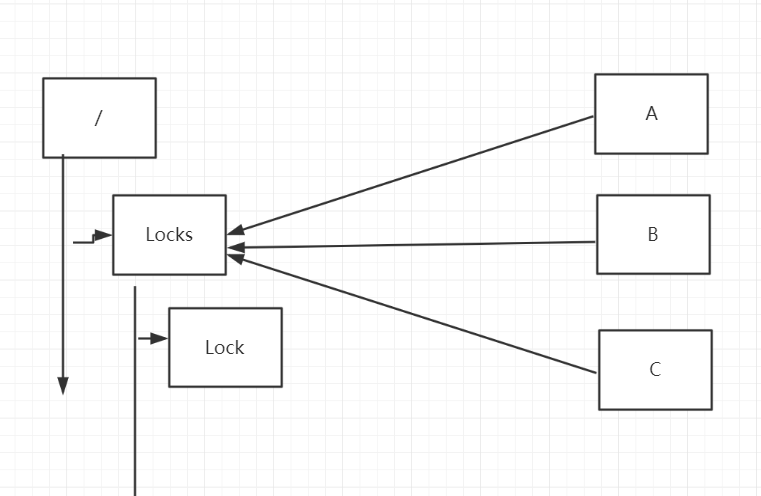
简明扼要的说: 现在在集群中我们如何知道分派去工作.首先通过选取leader 的方式生成一个领导,由这个领导去协调工作,同时告知其他的端口,现在这一块归我管理,有事找我.团体内部发生变化之后又会推选出一个领导,重复上诉操作.

### 分布式锁

分布式锁主要用于在分布式环境中保护跨进程、跨主机、跨网络的共享资源实现互斥访问，以达到保证数据的一致性。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

分布式锁设计原理图

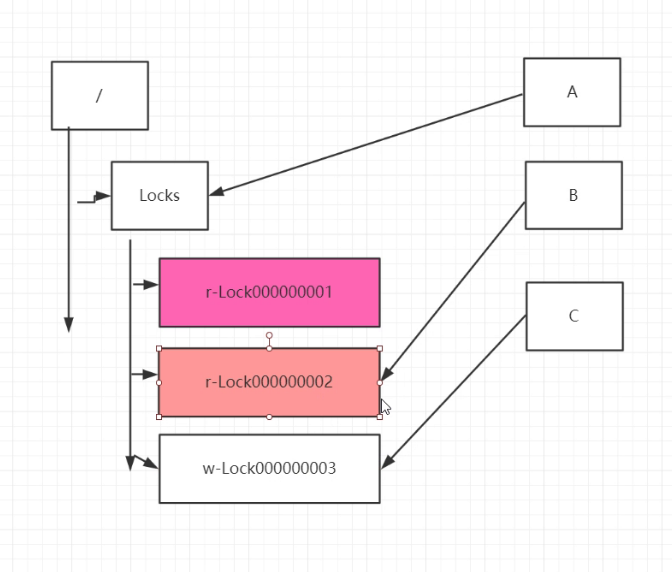
1、基于排它锁(保持独占)：



让所有的客户端（A B C）都在指定的Locks节点下 创建一个同名的节点lock，利用ZK 节点名称的唯一性来做限制， 谁创建成功了谁就获取到了锁，否则就是没有获取到。

这种设计比较简单

2、基于共享锁(控制时序)：



1、在当前节点下创建临时有序节点C，若创建成功则会返回节点C的名称。

2、获取当前节点下的所有临时节点。利用zk 节点的 顺序一致性，获取到了最小的节点就算获取到了锁。

3、找到临时节点中的最小节点的名称，和第一步创建节点返回的 C节点名称进行比较，如果匹配成功，表示当前节点就是最小节点，就意味着获取到了锁。

否则就要获取到比刚刚创建的节点顺序小的节点。

4、然后获取到比当前节点顺序小的所有节点的一个集合。如果不为空，就获取这个集合节点中的最后一个节点B。然后让B去监听它的上一个A，A的会话超时或者A节点被删除（释放）了，B就获取到了锁，同理，当B被删除或者会话超时时，C就获取到了锁。

5、执行完业务逻辑操作后 执行 删除节点的方法 锁就被释放了，然后下个节点就能获取到锁了。

场景说明:

1客户端连接zookeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-0000000000，第二个为/lock/lock-0000000001，以此类推。

2客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听/lock的子节点变更消息，获得子节点变更通知后重复此步骤直至获得锁；

3执行业务代码；

4完成业务流程后，删除对应的子节点释放锁。

步骤1中创建的临时节点能够保证在故障的情况下锁也能被释放，考虑这么个场景：假如客户端a当前创建的子节点为序号最小的节点，获得锁之后客户端所在机器宕机了，客户端没有主动删除子节点；如果创建的是永久的节点，那么这个锁永远不会释放，导致死锁；由于创建的是临时节点，客户端宕机后，过了一定时间zookeeper没有收到客户端的心跳包判断会话失效，将临时节点删除从而释放锁。

另外细心的朋友可能会想到，在步骤2中获取子节点列表与设置监听这两步操作的原子性问题，考虑这么个场景：客户端a对应子节点为/lock/lock-0000000000，客户端b对应子节点为/lock/lock-0000000001，客户端b获取子节点列表时发现自己不是序号最小的，但是在设置监听器前客户端a完成业务流程删除了子节点/lock/lock-0000000000，客户端b设置的监听器岂不是丢失了这个事件从而导致永远等待了？这个问题不存在的。因为zookeeper提供的API中设置监听器的操作与读操作是原子执行的，也就是说在读子节点列表时同时设置监听器，保证不会丢失事件。

最后，对于这个算法有个极大的优化点：假如当前有1000个节点在等待锁，如果获得锁的客户端释放锁时，这1000个客户端都会被唤醒，这种情况称为“羊群效应”；在这种羊群效应中，zookeeper需要通知1000个客户端，这会阻塞其他的操作，最好的情况应该只唤醒新的最小节点对应的客户端。应该怎么做呢？在设置事件监听时，每个客户端应该对刚好在它之前的子节点设置事件监听，例如子节点列表为/lock/lock-0000000000、/lock/lock-0000000001、/lock/lock-0000000002，序号为1的客户端监听序号为0的子节点删除消息，序号为2的监听序号为1的子节点删除消息。

从我的理解上来说,分布式锁就是为了保障当前的请求能够独立不受外界干扰.通过临时顺序节点所依赖的分布式锁机制,可以在高并发的情况下,将客户端请求按序号进行执行,其他的就要等待.一旦这种临时序列节点的端上的关闭了,就让下一个客户端获取最小序号从而执行后续操作.循环下去.

### 队列管理

两种类型的队列：

1、 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。

同步队列,

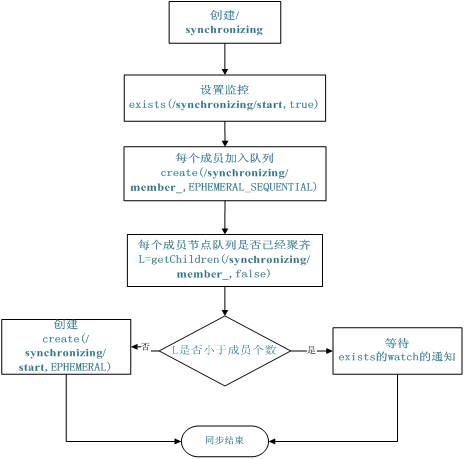
1. 当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。
2. 队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

同步队列用 Zookeeper 实现的实现思路如下：

创建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方式就是创建 /synchronizing/member\_i 的临时目录节点，然后每个成员获取 / synchronizing 目录的所有目录节点，也就是 member\_i。判断 i 的值是否已经是成员的个数，如果小于成员个数等待 /synchronizing/start 的出现，如果已经相等就创建 /synchronizing/start。

用下面的流程图更容易理解：

图 5同步队列流程图



先进先出队列

在zookeeper中先创建一个根目录 queue\_fifo，做为队列。入队操作就是在queue\_fifo下创建自增序的子节点，并把数据放入节点内。出队操作就是先找到queue\_fifo下序号最下的那个节点，取出数据，然后删除此节点。

### Zookeeper 监视（Watches） 简介

Zookeeper C API 的声明和描述在 include/zookeeper.h 中可以找到，另外大部分的 Zookeeper C API 常量、结构体声明也在 zookeeper.h 中，如果如果你在使用 C API 是遇到不明白的地方，最好看看 zookeeper.h，或者自己使用 doxygen 生成 Zookeeper C API 的帮助文档。

Zookeeper 中最有特色且最不容易理解的是监视(Watches)。Zookeeper 所有的读操作——getData(), getChildren(), 和 exists() 都 可以设置监视(watch)，监视事件可以理解为一次性的触发器， 官方定义如下： a watch event is one-time trigger, sent to the client that set the watch, which occurs when the data for which the watch was set changes。对此需要作出如下理解：

（一次性触发）One-time trigger

当设置监视的数据发生改变时，该监视事件会被发送到客户端，例如，如果客户端调用了 getData("/znode1", true) 并且稍后 /znode1 节点上的数据发生了改变或者被删除了，客户端将会获取到 /znode1 发生变化的监视事件，而如果 /znode1 再一次发生了变化，除非客户端再次对 /znode1 设置监视，否则客户端不会收到事件通知。

（发送至客户端）Sent to the client

Zookeeper 客户端和服务端是通过 socket 进行通信的，由于网络存在故障，所以监视事件很有可能不会成功地到达客户端，监视事件是异步发送至监视者的，Zookeeper 本身提供了保序性(ordering guarantee)：即客户端只有首先看到了监视事件后，才会感知到它所设置监视的 znode 发生了变化(a client will never see a change for which it has set a watch until it first sees the watch event). 网络延迟或者其他因素可能导致不同的客户端在不同的时刻感知某一监视事件，但是不同的客户端所看到的一切具有一致的顺序。

（被设置 watch 的数据）The data for which the watch was set

这意味着 znode 节点本身具有不同的改变方式。你也可以想象 Zookeeper 维护了两条监视链表：数据监视和子节点监视(data watches and child watches) getData() and exists() 设置数据监视，getChildren() 设置子节点监视。 或者，你也可以想象 Zookeeper 设置的不同监视返回不同的数据，getData() 和 exists() 返回 znode 节点的相关信息，而 getChildren() 返回子节点列表。因此， setData() 会触发设置在某一节点上所设置的数据监视(假定数据设置成功)，而一次成功的 create() 操作则会出发当前节点上所设置的数据监视以及父节点的子节点监视。一次成功的 delete() 操作将会触发当前节点的数据监视和子节点监视事件，同时也会触发该节点父节点的child watch。

Zookeeper 中的监视是轻量级的，因此容易设置、维护和分发。当客户端与 Zookeeper 服务器端失去联系时，客户端并不会收到监视事件的通知，只有当客户端重新连接后，若在必要的情况下，以前注册的监视会重新被注册并触发，对于开发人员来说 这通常是透明的。只有一种情况会导致监视事件的丢失，即：通过 exists() 设置了某个 znode 节点的监视，但是如果某个客户端在此 znode 节点被创建和删除的时间间隔内与 zookeeper 服务器失去了联系，该客户端即使稍后重新连接 zookeeper服务器后也得不到事件通知。

文中若有不理解推荐先看下Paxos算法.

仍有不理解可以参考

参考

分布式服务框架 Zookeeper -- 管理分布式环境中的数据

<https://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/>

ZooKeeper 工作、选举 原理

<https://blog.csdn.net/liuj2511981/article/details/42460069>

基于Zookeeper的分布式锁

<https://blog.csdn.net/qiangcuo6087/article/details/79067136>

如何用zookeeper 实现分布式锁

<https://www.2cto.com/kf/201804/736541.html>

<https://www.cnblogs.com/skying555/p/7873345.html>